

Quoted Staged Rewriting: A Practical Approach to Library-Defined Optimizations

Lionel Parreaux, Amir Shaikhha, Christoph Koch
EPFL, Switzerland — {firstname.lastname}@epfl.ch

Abstract

Staging has proved a successful technique for programmatically removing code abstractions, thereby allowing for faster program execution while retaining a high-level interface for the programmer. Unfortunately, techniques based on staging suffer from a number of problems — ranging from practicalities to fundamental limitations — which have prevented their widespread adoption. We introduce *Quoted Staged Rewriting* (QSR), an approach that uses type-safe, pattern matching-enabled quasiquotes to define optimizations. The approach is “staged” in two ways: first, rewrite rules can execute arbitrary code during pattern matching and code reconstruction, leveraging the power and flexibility of staging; second, library designers can orchestrate the application of successive rewriting phases (stages). The advantages of using quasiquote-based rewriting are that library designers never have to deal directly with the intermediate representation (IR), and that it allows for non-intrusive optimizations — in contrast with staging, it is not necessary to adapt the entire library and user programs to accommodate optimizations.

We show how Squid, a Scala macro-based framework, enables QSR and renders library-defined optimizations more practical than ever before: library designers write domain-specific optimizers that users invoke transparently on delimited portions of their code base. As a motivating example we describe an implementation of stream fusion (a well-known deforestation technique) that is both simpler and more powerful than the state of the art, and can readily be used by Scala programmers with no knowledge of metaprogramming.

CCS Concepts • **Software and its engineering** → **Software performance**; Compilers;

Keywords Rewrite Rules, Staging, Optimization

ACM Reference Format:

Lionel Parreaux, Amir Shaikhha, Christoph Koch. 2017. Quoted Staged Rewriting: A Practical Approach to Library-Defined Optimizations. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE’17)*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3136040.3136043>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GPCE’17, October 23–24, 2017, Vancouver, Canada

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-5524-7/17/10...\$15.00

<https://doi.org/10.1145/3136040.3136043>

1 Introduction

We begin by providing some necessary background before presenting our approach and core contributions.

1.1 Staging and extensible compilers

Multi-stage programming (MSP, or just *staging*) [46] lets programmers syntactically distinguish multiple stages of execution in their programs. At each intermediate stage the program computes away what is known at this stage, and generates a new *residual* program meant to execute the next stage. The ultimate stage performs the task of the unstaged program, but in a more efficient way. MSP can be viewed as a form of partial evaluation with explicit annotations for binding-time analysis [19], or as a way to define type-safe program generators that work by iteratively composing code fragments together. MSP can be applied to both run time [45, 47] and static [13, 56] code generation. In the latter case, programs are made of two stages where the first stage is executed at compile time and the second stage corresponds to the final, compiled program. A major limitation of MSP is that it generally offers no type-safe facilities to analyse code (it is *purely generative*), which greatly restricts its capabilities in terms of program optimization. Moreover, staging a library exposes users of that library to staging annotations,¹ which leak through its interface. Perhaps more importantly, staging requires to decide from the beginning which parts of the program are static (meant to be executed at program generation time) and which parts are dynamic, then building everything around that dichotomy, making it hard to evolve the design later on without extensive refactorings.

Staging has been successfully applied to optimizing domain specific languages (DSL), especially Embedded DSLs (EDSL) [17] which are DSLs that are defined within a more expressive *host language* such as Haskell [1, 17, 29] or Scala [26, 30, 35, 39]. In this context, staging has been used to facilitate the definition of extensible compilers for performance-oriented DSLs and heterogeneous target platforms [1, 9, 26, 30, 33]. Generally speaking, these compilers reuse the front-end capabilities of their host (syntax and type system) but they convert programs into their own domain-specific intermediate representation (IR) before stringifying low-level code. This limits their ability to interact with code outside of the DSL. For example, a compiler for a streams DSL (see the LMS embedding of [24]) by default can only handle primitive types, strings, arrays, functions, loops and tuples, and adding support for using other constructs (such as, for example, `BigInt`) requires extending the compiler’s IR, which

¹ Type-Based Embedding eschews staging quotations [35], but requires more complex types, which also degrades the library interface [23, 34].

involves significant amounts of boilerplate. Moreover, expressing non-trivial optimizations in these frameworks is hard and error-prone, as one has to deal with details of the IR with limited support for code pattern matching. Tools have been proposed to generate some of the boilerplate automatically [23, 43] and solutions were sketched to make code rewriting easier [34], but the fundamental limitations and intrinsic complexity of these approaches are still there, and the burden they impose on library users only partly lifted.

1.2 User-defined rewriting

The idea of building program optimizations via high-level rewrite rules is far from new [10, 21, 25, 33, 41, 42, 49–51]. However, few approaches have offered a lightweight, type-safe, language-integrated way of expressing these rules, as most rely on distinct specification metalanguages or complex code transformation combinators.

A notable exception, the Glasgow Haskell Compiler (GHC) [21] allows library writers to describe simple algebraic rewrite rules consisting of two expressions: a pattern, and a template to replace the pattern with when the rule fires. GHC tries to apply as many of these rules as possible as it performs its own optimization passes. There are no termination or correctness guarantees associated with the rewrite rules, as their objective is to let users make — at their own risk — domain-specific assumptions that the compiler cannot make.

There are two major limitations to this approach. First, while the rules are sufficient to express a variety of optimizations, they are limited in the patterns that they can match and in the code that they can generate. For example, it is easy to define a rule to rewrite `pow x 2` into `x * x`, but the generalization of that rule to rewrite `pow x n`, where `n` is constant, into `x * ... * x` is not expressible. Second, library designers have very weak guarantees about the actual application of their rules when a program is compiled. The result is intimately dependent on the inlining behavior of GHC (which is affected by separate compilation), so that expert knowledge about the inner workings of the GHC optimizer is often required to achieve satisfying results [21].² As a consequence, the practice is to annotate functions with `INLINE` or `NOINLINE` directives that sometimes need to refer to GHC's own internal optimization phase numbers. Moreover, approaches like stream fusion — a popular deforestation technique [6, 7] — have been shown to require more powerful rewriting facilities than simple GHC rewrite rules. This has sparked interest in HERMIT [10, 11], an interactive system based on rewriting combinators that is significantly more complex.

1.3 Quoted Staged Rewriting

In summary, staging is powerful but imposes a burden on both library users and library designers. Moreover, purely-generative staging disallows code inspection, which is limiting, and approaches that allow code inspection do so by exposing low-level IR constructs that are hard to manipulate.

² The GHC wiki has this informal bit about the behavior of rewrite rules in the context of list fusion: “Q: Why does making one thing fuse sometimes make something else not fuse? A: Because the whole system is built around inlining, and no one really knows how to make that Do The Right Thing every time. Also, no one knows a better way to avoid basing it on inlining.”

Rewrite rules in the style of GHC are easier to express and integrate more seamlessly with the host language, but they lack expressiveness and control.

In this paper, we bring together the advantages of staging and rewriting into a unified framework, *Quoted Staged Rewriting* (QSR), based on the Squid metaprogramming system [31]. We claim that our approach combines the flexibility and ease of language-integrated rewrite rules with the power and guarantees of static staging.

Our framework works by **user-defined, scoped optimizations**, whereby: 1. library designers express powerful domain-specific optimizations by way of type-safe quasiquote-based rewrite rules; and 2. library users write normal, un-staged code that they can surround with `optimize{ ... }` blocks in order to apply those library optimizations. With first-class control of inlining, users can abstract on the library's constructs while letting the library see through these abstractions to apply its rewritings. Scoped optimizations are useful because it often makes sense to focus optimization efforts on the “hot execution paths” of a program, where we can afford to let the optimizer spend more time doing its job. In our experience, applying these aggressive optimizations to more code outside of the hot paths makes the general compilation slower but has rapidly diminishing results.

Rewrite rules, which are applied at compile time, are allowed to use **arbitrary computations**, a flexibility that places them on equal footing with staging. On the other hand, rewriting enables a more dynamic approach to binding-time analysis. Together with extensible pattern matching, this favors more modular optimization designs: rewritings with orthogonal concerns can be completely decoupled. Taking inspiration from transformation-based compilers [20, 36], where rewritings are interspersed with successive **lowering phases** that decrease the general level of abstraction, we allow optimization designers to specify at which phase to inline which library abstractions. This is an essential feature to guarantee consistent abstraction removal and robust, predictable optimization — both staples of MSP. We make novel use of a simple IR and effect system, to soundly accommodate Scala's imperative features while enabling high-level algebraic rewritings. We thus reap the benefits of purity while still allowing the manipulation of low-level imperative programs.

1.4 Contributions

Our contributions are organized as follows:

- We develop a simple staging example, and show how the problem of extending it to handle more optimizations exhibits limitations with staging (Section 2).
- We show how QSR eliminates these problems while retaining the desirable properties of staging, and we see how to modularly define and compose QSR libraries (Section 3).
- We explain important aspects of how Squid is implemented and how its IR and effect system work to enable QSR in the presence of imperative features (Section 4).
- We detail our main use-case: implementing stream fusion by QSR.³ We improve on the state of the art by expressing the fusion of more programs, in a simpler way (Section 5).

³ Source code freely available at <http://github.com/LPTK/Squid-Examples>.

- We empirically verify that the QSR approach to stream fusion matches staging in terms of performance, but also that it requires less lines of code (Section 6).

2 Background on Staging

2.1 Preamble: Typed Quasiquotes

In this paper we will write all examples using our Squid framework, which can express staging as well as rewriting. The way Squid differentiates code in the current stage from code in the next stage is by the use of `code "..."` quasiquotes and associated `Code[...]` types. For example, while expression `2.toDouble` evaluates immediately to value `2.0` of type `Double`, expression `code"2.toDouble"` evaluates to a *code value* representing the program fragment `2.toDouble` of type `Code[Double]`. In order to go from a `Code[Double]` to a `Double`, one has to call `run` on the result, as in `code"2.toDouble".run` which evaluates to `2.0`.

Quasiquotes support composing code fragments together by using `$`-escapes, which represent “unquotes.” For example, `val d = code"2"; code"$d + 1"` evaluates to `code"2+1"`. It is possible to unquote the result of bigger expressions by using curly braces after the dollar sign, so the previous code can also be written `code"${ code"2" } + 1"`.

Although code quasiquotes use a syntax similar to string quotation `"..."`, remark that the code fragments they generate are *not* internally represented as character strings, but as constructs in Squid’s intermediate representation (IR).

2.2 Staging the Power function

The prototypical staging example is *power*, a function that raises a number x to the n^{th} power: since the exponent part n is often a statically known integer, it is tempting to specialize that function so that a call to it expands into a simple sequence of multiplications. Figure 1a presents the code for a *staged* power function, which takes a current-stage exponent n and returns a function from any `Double` code value base to a code value representing its multiplication n times. Triple quotation marks `"""` introduce multi-line quotations.

The figure ends with a usage example for $n = 3$. The astute reader will have noticed that the unquoted expression `power(3)` has type `Code[Double] => Code[Double]`, whereas it is applied in the next stage (inside the quotation) as if it were of type `Code[Double => Double]`. The reason is that Squid automatically lifts any current-stage function `Code[A] => Code[B]` into a next-stage function `Code[A => B]` upon insertion, and immediately inlines it.

2.3 New optimization opportunity

When removing abstractions programmatically and performing aggressive inlining, optimizable patterns often emerge, including patterns that a programmer would never write explicitly [21]. For instance consider a simulation application that needs to compute the gravitational force between several different celestial bodies. Those of us who remember our physics course will point out that the relevant equation has the form $F = G \frac{m_0 \cdot m_1}{d(p_0, p_1)^2}$, which corresponds to

the program of Figure 2. When the call to `distance` is inlined, the body of `gravityForce` ends up containing a call to `pow(sqrt(...), 2)`, which is clearly an inefficient identity:

```
G * plan0.mass * plan1.mass /
  pow(sqrt( pow(plan0.pos.x - plan1.pos.x, 2)
            + pow(plan0.pos.y - plan1.pos.y, 2) ), 2)
```

Naturally, we would like to be able to optimize such patterns.

2.4 Limitations of Staging

Let us consider what happens if we stage the function of Figure 2, making use of the `power` function defined in Figure 1a. Assuming purely-generative staging like in MetaOCaml [45], there is no easy way to extend that definition of `power` to perform the “power-of-power” optimization described above. The staged function can no longer accept a `Code[Double]` as the base, since purely generative approaches do not allow *inspecting* or *decomposing* code fragments – only *creating* and *composing* them together. However, we need to know whether a given piece of code has the form of a square root to be able to eliminate a square operation performed on it.

Solving this issue typically involves defining an auxiliary data structure for code being constructed, that carries additional information about its underlying structure. Figure 4 shows a generalized definition `genPower` of the power function, that uses an algebraic data type `CodeRep` to retain information about the code: subclass `Pow` describes a piece of code that results from an application of the power function, while `Simple` corresponds to other code. Method `toCode` is used to *reify* that intermediate representation into a proper code fragment to be inserted into a quasiquote.

Notice how that change affected the way we define `pow3`. More complex usages of `power` have to change in an even more significant way, as is shown in Figure 3 where we adapt the planet simulation code seen previously to our new staging scheme. As one can see, both the implementation of `genPower` and its usage in `pow3`, `distance` and `gravityForce` become tremendously more complicated. We believe that this is why purely-generative staging is often relegated to the back end – i.e., used merely for end-of-the-pipeline code generation, while the front end of the library is defined in the finally-tagless style [2] and mostly hides staging.

Extensible compiler techniques obviate the need to explicitly wrap and unwrap code, by making the equivalent of `Pow` directly extend (inherit from) the compiler’s internal IR node type [16]. Moreover, type-inference-based techniques help to hide staging annotations to some extent [35], which can be further improved by language virtualization [23, 28], but the added complexity and fundamental limitations are still there: DSL designers have to write the entire library with staging in mind, define IR nodes for all constructs meant to be supported by the DSL, and interact directly with details of the compiler’s IR (especially when defining rewritings [34]).

3 Quoted Staged Rewriting

3.1 Preamble: Quasiquotes in Pattern Matching

Squid supports *pattern matching* on code fragments, with syntax `case code"pattern" => result`. In a *code* pattern, the semantics of unquotes is no longer to *insert* but rather to


```
def power(exp: Int)(base: Code[Double]): Code[Double] =
  if (exp == 0) code"1.0"
  else {
    assert(exp > 0)
    if (exp % 2 == 0) code""
      val tmp = ${power(exp/2)(base)}
      tmp * tmp
    ""
    else code"$base * ${power(exp-1)(base)}" }
val pow3 = code"(x: Double) => ${power(3)}(x)".run
```

(a) Defining a staged power function.

```
import Math.pow // (Double, Double) => Double
@bottomUp @fixedPoint val powOpt = rewrite {
  case code"pow($base, 0)" => code"1.0"
  case code"pow($base, ${Const(exp)})"
    if exp.isWhole && exp > 0 =>
      if (exp % 2 == 0) code""
        val tmp = pow($base, ${Const(exp/2)})
        tmp * tmp
      ""
      else code"$base * pow($base, ${Const(exp-1)})" }
def pow3(x: Double) = powOpt.optimize { pow(x, 3) }
```

(b) Rewriting the standard Math.pow function.

Figure 1. Two approaches to optimizing the power function.

```
import Math.{pow, sqrt}; val G = 6.67E-11
def gravityForce(plan0: Planet, plan1: Planet) =
  G * plan0.mass * plan1.mass /
    pow(distance(plan0.pos, plan1.pos), 2)
def distance(p0: Position, p1: Position) =
  sqrt(pow(p0.x - p1.x, 2) + pow(p0.y - p1.y, 2))
```

Figure 2. Example simulation code using sqrt and pow.

```
def gravityForce(p0: Code[Planet], p1: Code[Planet]) =
  code""G * $p0.mass * $p1.mass / ${
    genPower(distance(code"$p0.pos", code"$p1.pos"), 2.0)
    .toCode }""
def distance(p0: Code[Position], p1: Code[Position]) =
  sqrt(Simple(code""
    ${genPower(Simple(code"$p0.x - $p1.x"), 2.0).toCode}
    + ${genPower(Simple(code"$p0.y - $p1.y"), 2.0).toCode}
    """))
def sqrt(x: CodeRep[Double]) = genPower(x, 0.5)
```

Figure 3. Simulation code adapted for the (new) staged interface.

```
abstract class CodeRep[T] { def toCode: Code[T] }
case class Simple[T](toCode: Code[T]) extends CodeRep[T]
case class Pow(cde: Code[Double], exp: Double)
  extends CodeRep[Double] { def toCode =
    if (exp.isWhole) power(exp.toInt)(cde)
    else code"Math.pow($cde, ${Const(exp)})" }
def genPower(base: CodeRep[Double], exp: Double) = base match {
  case Simple(c) => Pow(c, exp)
  case Pow(c, e) if exp.isWhole => Pow(c, exp * e)
  case _ => Pow(base.toCode, exp) }
val pow3 = code"(x: Double) => { (x: Code[Double]) =>
  genPower(Simple(x), 3).toCode }(x)".run
```

Figure 4. New definition of the staged power function, with support for optimizing the “power-of-power” pattern.

extract code fragments found in the place where they occur. For example, the following program evaluates to `code"2"`:

```
code"2+1" match { case code"($n: Int) + 1" => n }
```

3.2 Rewriting Math.pow

Figure 1b presents a rewriting that optimizes calls to `Math.pow` with integer exponents using binary exponentiation. `Const` is the constructor/extractor for constant values; for example `Const(2)` is equivalent to `code"2"` in both expressions and patterns. A rewriting is registered using a `rewrite` block containing pattern matching clauses. Each rewriting can be configured to apply in bottom-up or top-down traversal order, and can be made to apply repeatedly until a fixed

point is reached. In this example we use bottom-up order and fixed-point recursion. Method `isWhole` simply tests whether a `Double` value is a whole number. Note that Squid uses an IR based on the A-Normal Form (ANF) [12], which means that non-trivial sub-expressions are let-bound, so that it is not a problem to duplicate the base argument extracted from the patterns. For example, `pow(readInt, 3)` is rewritten into:

```
val x_0 = readInt; x_0 * 1.0 * (x_0 * 1.0 * (x_0 * 1.0))
```

One can immediately notice several differences with the staged version. First we do not need to create a new, distinct power construct; instead we operate directly on Java’s standard `Math.pow` method. This means that any programs using `Math.pow` can already benefit from our optimization without any changes to their business logic. In other words, QSR allows us to work directly on program representations instead of staged structures, but without having to define our own domain-specific IR. Moreover, the optimization of `pow3` in Figure 1b happens at compile-time which makes the user experience similar to built-in optimizations.

3.3 Extending the Rewriting

Implementing the “power-of-power” optimization by rewriting is straightforward, as we can simply add the following rewrite rules⁴ to those of Figure 1b:

```
case code"sqrt($x)" => code"pow($x, 0.5)"
case code"pow(pow($base, ${Const(a)}), ${Const(b)})"
  if b.isWhole => code"pow($base, ${Const(a * b)})"
case code"pow($x, 1)" => x // just for aesthetics
```

We can now wrap the body of `gravityForce` in Figure 2 inside a `powOpt.optimize{...}` block, which rewrites it into:

```
val x_0 = plan0.pos.x - plan1.pos.x
val x_2 = plan0.pos.y - plan1.pos.y
G * plan0.mass * plan1.mass / (x_0 * x_0 + x_2 * x_2)
```

There is one caveat however: the additional rules have to be inserted *at the beginning* of the `case` clauses of Figure 1b, otherwise an expression such as `pow(pow(x, 0.5), 2)` will be rewritten to `val tmp = pow(y, 0.5); tmp * tmp` by the second rule of Figure 1b, before the new rules can be applied, missing that optimization opportunity.⁵ This shows

⁴ We require the outer exponent `b` to be a whole number to avoid performing unsound reductions, like `sqrt(pow(x, 2))` to `x` instead of `abs(x)`.

⁵ The cases of a rewriting are tried in the order they are defined, similar to classical pattern matching (the `match` keyword).

```
@online val powOpt = rewrite {
  case code"pow($base, ${Const(exp)})"
    if exp.isWhole && exp > 0 => power(exp.toInt)(base)
  case code"pow($base, $exp)" => throw
    StagingError("Non-static exponent: " + exp) }
```

Figure 5. Hybrid approach: rewrite rules that falls back to staged function and emit error on rewrite failure.

that the ordering of rewritings should be carefully considered by library designers. More generally, it is often useful to organize rewrite rules into separate phases. For example, considering that the implementation of `sqrt` is faster than that of more general-purpose `pow`, it would pay off to have a later phase that converts code of the form `pow(x, 0.5)` back into `sqrt(x)` before emitting the final code. The question of rewriting phases is exemplified further in Section 5.4.

3.4 Hybrid approaches and online rewriting

It is possible to freely combine rewriting and traditional staging. For instance, while fixed point rewriting is often useful, its use in Figure 1b could be considered overkill; instead of looping through the fixed point of the rewrite rule, we could just as well call a staged function that performs the looping itself,⁶ as demonstrated in Figure 5. In that configuration, the role of rewrite rules is to automatically extract static parts from unannotated programs, similar to binding time analysis (BTA). Reminiscent of online partial evaluation [19], the `@online` annotation specifies that a rewriting should be performed on the fly, as program representations are constructed. Online rewrite rules can be used to achieve a form of normalization: by restricting the space of representable programs, they make transformations simpler to express. Additionally, they can alleviate phase ordering problems [36].

3.5 Guarantees and Control

Thanks to arbitrary code execution in rewrite rules and control over inlining (see Section 5.4), we make the argument that QSR is as powerful as compile-time staging. In particular, it preserves all the necessary control required by library designers, who wish to guarantee to library users that program optimizations apply reliably: if some rewriting could not be applied because static information could not be extracted, it is always possible for the rewrite system to emit a compile-time error and fail code generation, as is done in the second rule of Figure 5. Other valid behaviors in this case may be: emitting a compile-time warning but going through with code generation; simply logging the failure in a report that users can inspect in order to understand how to speed up their program; or doing nothing at all – which is what traditional compiler optimizations have been doing.

An even stronger argument can be made following [3] and [29], who rely on the subformula principle of normal proofs adapted to programming [54] to guarantee that types that do not appear as subformulas of the types of the inputs and outputs of a program will be completely removed after sufficient normalization. For example, a program of type

⁶ Yet another hybrid approach would be to use code pattern-matching inside of a staged definition, lifting the purely-generative restriction.

`Int => Int` that is internally defined using some `Stream` data type can be rewritten to a program that does not make use of `Stream`, as long as we can inline all `Stream` functions. The possibility of inlining the relevant functions is an integral part of the subformula principle: if we do not have access to the function body (and therefore cannot inline it), the function itself should be counted as part of the inputs to the program, which prevents the application of the subformula principle (as the function type will contain the offending type – here, `Stream`).

3.6 Modularity of rewritings

Squid enables the common approach [41, 52] of separating term-level rewritings from transformation strategies. As a result, it is possible to define self-contained libraries of useful rewritings as well as libraries of useful transformation strategies and compose them modularly. To combine different strategies we rely on Scala’s mix-in composition mechanism, a technique also used in previous work [16, 37].

Another important direction for modularity is to allow abstracting over patterns in rewrite rules [49]. Squid achieve this by merely relying on Scala’s built-in custom extractors:

```
object Even { def unapply(x: Code[Double]) = x match {
  case Const(n) if n % 2 == 0 => Some(x)
  case code"($_: Int) * 2"    => Some(x)
  case code"$Odd(_) + 1"     => Some(x)
  case _                      => None
}
object Odd { /* similar definition elided */
  def unapply(x: Code[Double]) = x match {
  case code"pow(-1, ${Even(n)})" => code"1.0"
  case code"pow(pow($b, ${Even(Const(n))}), ${Const(e)})"
    if n * e == 1.0 => code"abs($b)" }
```

The code above defines co-recursive `Even` and `Odd` extractors that are used to rewrite terms such as `pow(pow(x, 2), 0.5)` into `abs(x)` and `pow(-1, readInt*2)` into `readInt; 1.0`.

3.7 Composing uses of QSR libraries

Finally, we describe how to compose together optimizers defined in different libraries. The simplest way to achieve composition is to *nest* the optimize blocks, which expand inside-out: in `r0.optimize{ ... r1.optimize{ ... } ... }` the `r1` block expands first (applying its rewrite rules), and what the `r0` block sees is the resulting optimized code. Consequently, this approach may yield different results depending on the order in which the different blocks are nested.

A more fine-grained alternative is to merge rewriting passes together, as in `(r0+r1).optimize{ ... }` but this requires that the rewritings be defined using compatible traversal strategies. More advanced library optimizers (like in Section 5) may be defined in terms of successive rewriting and inlining phases; determining how to mix these more complex optimizers together in a fine-grained way requires careful consideration from the user.

3.8 Optimizing Existing Libraries

As we saw with `Math.pow` and `Math.sqrt`, Squid can optimize code written using preexisting, unmodified libraries. On the other hand, it is often beneficial to design libraries

with optimization in mind, using constructs that can be easily manipulated and optimized by code rewriting. For example, Section 5 presents a streams implementation `Strm` that is geared towards QSR. Nevertheless, it is still possible to use that ad-hoc implementation to optimize existing libraries, such as Scala's `Stream`. This is done in three phases. First, we define conversion functions `toStrm` and `toStream` to move between the two representations [48]. Then we rewrite all `Stream` operations to `Strm` ones using these conversions, while collapsing useless conversions on the fly. For example, we convert `Stream.from(0,1).take(3).sum` to `Strm.from(0,1).take(3).sum` with these rewritings:

```
case code"($xs:Stream[Int]).sum" => code"toStrm($xs).sum"
case code"Stream.from($start,$step)"
  => code"toStream(Strm.from($start,$step))"
case code"($xs:Stream[Int]).take($n)"
  => code"toStream(toStrm($xs).take($n))"
case code"toStrm(toStream($xs:Strm[$t]))" => xs
```

Finally, the usual `Strm` optimizations can be applied on the resulting program, producing optimized code that may entirely bypass the usage of `Stream`. Note that in certain cases, inserting back-and-forth conversions may be *detrimental* to performance when the whole pipeline cannot be converted and when the cost of conversion outweighs the gains of optimization [7]. Thankfully, it is easy to write a separate “clean-up” phase which reverts conversions that could not apply fully, avoiding unwanted conversion costs.

4 Enabling Quoted Staged Rewriting

In this section, we detail several important technical aspects of the Squid implementation that enable QSR.

4.1 Custom A-Normal Form (ANF)

Hash consing. The Squid ANF IR is geared towards making rewrite rules as flexible as possible. As such, we have an unconventional definition of “non-trivial expressions” (those expressions that need to be let-bound). In our approach all pure expressions are considered trivial and therefore they are never let-bound. Semantically, it is *as if* pure expressions were duplicated at every one of their use sites, but internally Squid uses hash-consing [18] so that there is only a single representation in memory of every pure term. Conceptually, writing `code"println(x+1); x+1"` is equivalent to writing `val x_0 = code"x+1"; code"println($x_0); $x_0"`. This is not only useful to save memory, but also allows Squid to cache transformations so that they are not performed more than once on a given pure term.

Matching. The mechanism described above gives us the benefits of ANF (normalized control-flow, sound handling of effects) while enabling powerful code pattern matching, because patterns can freely inspect nested sub-expressions as long as these sub-expressions are pure. Impure patterns can also be used, such as `case code"println(readInt)"`, which matches `code"val x = readInt; println(x)"` but does not match any program where there are impure expressions intervening between the `readInt` and `println` calls, like `code"val x = readInt; readDouble; println(x)"`.

Scheduling. When generating or pretty-printing code, Squid uses a *scheduling* phase to let-bind pure expressions that are used multiple times, in order to minimize program size and computation costs. This phase needs special care around closures, by-name arguments and branching constructs. For example, it makes sense to schedule expressions *out* of a loop (so as not to recompute them on every iteration), but *inside* an if-then-else branch if the other branch does not also use it (so as not to perform useless computations). Due to the lack of space, we do not describe the algorithm used by Squid to perform scheduling, but there is extensive literature on the subject [4]. Squid allows users to annotate higher-order method parameters to indicate whether they are expected to execute *at most once*, *at least once* or *many times*. This way Squid can produce sensible schedules for code that uses custom constructs, such as the `loopWhile` function of Section 5.3.

4.2 Effect System

Basic Principles. In order to determine which expressions are pure, Squid uses a very simple yet surprisingly versatile effect system. The idea is to differentiate two kinds of effects: *direct* and *latent*. An expression has direct effect if it reads or writes mutable state, performs I/O, etc. Latent effects are reserved for expressions that *delay* the execution of direct effects, such as a lambda expression containing effectful code. Similarly to previous systems [38], methods are then annotated with 1. their intrinsic effect, and 2. the way they propagate the effects of their arguments. “Pure expressions” are those with no direct effect, so lambda expressions are considered pure even when they have latent effect.

Examples. Since the Scala `Stream` datatype is purely functional, none of its methods has any intrinsic effects. However, transformers like `map` “build up” latent effects when applied to effectful functions, so `Stream(1,2,3).map(_ + readInt)` has latent effect. On the other hand, consumers like `fold` “execute” the latent effect of their arguments. This is because `Stream` is a *lazy* data structure that executes computations only when required. As an example, if either `s` or `f` have latent effect then `s.fold(0)(f)` has direct effect — otherwise it is pure. As a result, a stream pipeline like the ones we study in Section 5 is normalized to one big expression terminated by a call to a consumer such as `fold` or `foreach`, which allows for simple yet powerful rewritings (cf. Figure 7). Finally, notice that *strict* collections behave differently: for them, transformers execute immediately, so code like `List(1,2,3).map(_ + readInt)` has direct effect.

Future Work. Improving the effect system to be more finely-grained could benefit pattern matching and scheduling. In particular, we could maintain an arbitrary number of latent effect layers — currently we consider that `x => print(x)` and `x => y => print(x)` have the same (latent) effect, which means that when the latter is applied *once* it is already considered to have a direct effect, which is not unsound but rather imprecise. We could also use a graph-based IR [5] to maintain explicit dependencies between expressions, like in LMS [36] or Graal [55]. Finally, while we currently require users to

annotate the effects of their methods, automatic effect inference is entirely feasible. Existing dedicated effect-tracking tools could also be leveraged, such as Scala FX [38].

4.3 Implementation of code pattern matching

Squid implements pattern matching by building a term containing holes to represent the pattern, similarly to the *Folds* subsystem of HERMIT described in Farmer’s dissertation (p.73) [10]. Squid’s current rewriting algorithm works by trying each pattern one after the other, and does not memoize previous matching results, although that would be possible to implement — following Farmer, we plan to use trie maps in order to speed up the process. Nevertheless, we have found that even our non-optimal approach was practical, and enabled advanced rewritings like those of Section 5 plus dozens of online normalization rules⁷ to be applied on mid-sized method bodies without incurring unmanageable compilation times. We reserve an empirical study of these performance characteristics for future work.

5 Stream Fusion

5.1 Previous Approaches

List fusion, which consists in optimizing libraries that make use of functional lists — or “streams” — by removing intermediate results (a technique also called *deforestation* [53]), has been an intense subject of research [7, 14, 15, 21, 22, 24]. Promising approaches relying on GHC rewrite rules [7, 21] were thoroughly explored, but these approach often suffer from a lack of control and from limitations of the GHC rewrite rule framework.

On the other hand, staging has been used to achieve similar goals [22, 24]. Most recently, Kiselyov et al. [24] demonstrated an approach based on staging that fuses several difficult stream operations, including “zipping” two streams together.⁸ Their approach requires an elaborate staged representation of streaming code, that relies on existentially-quantified types to encode the stream’s internal state and uses continuation-passing style (CPS) thoroughly to thread state and iteration code through the streaming constructs. This makes the description and implementation of the library slightly convoluted and hard to understand. On the other hand, the library exposes staging annotations, as it forces users to write all the business logic of their application inside quotations.⁹ For example, instead of writing `stream.map(x => x.foo)`, one has to write the equivalent of `stream.map(x => code "$x.foo")`. This has two disadvantages: first, library users generally have to use a compiler modified for staging, and need to understand staging annotations even when it’s irrelevant for their business logic; second, this means the BTA of the library is completely

fixed [27]; i.e., which parts are known statically is fully determined in advance by the library designers. Future changes to enable more optimizations may break the library interface, as we describe further in Section 5.6, and any usages of the library in a slightly more dynamic setting are impossible.

In the rest of this section, we show how to use QSR and reap the benefits of staging and rewriting: we perform stream fusion without affecting the user interface of the library and more thoroughly than in the staging-based previous work, and we enable more control and more powerful transformations than offered by GHC rewrite rules.

5.2 Stream Fusion by CPS and Inlining

The streams interface that we focus on is the same as in [24] and follows. All functions are standard and self-describing:

```
type Strm[A] <: {
  def map[B](f: A => B): Strm[B]
  def flatMap[B](f: A => Strm[B]): Strm[B]
  def take(n: Int): Strm[A]
  def filter(p: A => Boolean): Strm[A]
  def zipWith[B](that: Strm[B]): Strm[(A,B)]
  def fold[B](z: B)(f: (B,A) => B): B
}
def fromRange(from: Int, until: Int): Strm[Int]
def unfold[A,B](init:B)(next:B => Option[(A,B)]): Strm[A]
```

Other constructs can of course be defined on top of these ones by adding “syntactic sugar,” such as `fromArray` in:

```
def fromArray[A](xs: Array[A]): Strm[A] =
  fromRange(0, xs.length).map(i => xs(i))
```

A simple way to implement streams is by backing them with imperative *producers* which allow requesting elements one by one (the *pull* model of iteration) while keeping internal iteration state. In order to retain the expected pure interface for streams, it is necessary that stream objects not store a specific producer, but rather a way to initialize a new producer and its internal state — a producer factory:

```
case class Strm[A](producer: () => Producer[A])
```

`Producer[A]` can be implemented as a function of no arguments that, when called, returns `Some(e)` if `e` is the next element to be produced, or `None` if there are no more elements to produce (i.e., `type Producer[A] = () => Option[A]`). However, as has been noted before [44] the use of `Option` to guide control flow tends to generate code that is not easily optimized or partially evaluated, because it typically contains redundant branching expressions. In general, rewriting these into a more streamlined control flow requires some control-flow analysis. While this can certainly be done using our approach (since we can inspect code by recursively traversing it via pattern-matching), it is much easier to adopt an alternative representation of producers. As often, the better representation is in continuation-passing style:

```
type Consumer[A] = A => Unit
type Producer[A] = Consumer[A] => Unit
```

Figure 6 shows the gist of the `Strm` implementation that we finally settle on. The `andThen` combinator pipelines two functions together such that `f.andThen(g)` (also written in operator syntax `f andThen g`) is equivalent to `x => g(f(x))`.

⁷ When we did the microbenchmarks of Section 6, there were a total of 66 online rewrite rules registered, handling things ranging from logic operations to options normalization to desugaring common Scala idioms.

⁸ Zipping a stream of elements of type `A` with a stream of elements of type `B` produces a stream of combined `(A,B)` elements.

⁹ The authors propose to use combinators to hide staging and mitigate the issue, but we believe that this is not really helping. `x => stagedFoo(x)` is not qualitatively better for the user than the quoted version.

```
@embed case class Strm[A](producer: () => Producer[A]) {
  def map[B](f: A => B): Strm[B] = Strm(() => {
    val p = producer(); k => p(f andThen k) })
  def take(n: Int): Strm[A] = Strm(() => {
    val p = producer(); var taken = 0
    k => if (taken < n) { taken += 1; p(k) } })
  def zip[B](that: Strm[B]): Strm[(A,B)] = Strm(() => {
    val p0 = producer(); val p1 = that.producer()
    k => p0 { a => p1 { b => k((a,b)) } } })
  def fold[B](z: B)(f: (B,A) => B): B = {
    val p = producer(); var cur = z; var cont = true
    while (cont) { cont = false
      p { a => cur = f(cur, a); cont = true } }; cur }
  def foreach(f: A=>Unit): Unit = fold(())(_ , a => f(a))
  /* other implementations elided */
}
def fromRange(n: Int, m: Int): Strm[Int] = Strm(() => {
  var i = n; k => { if (i < m) { k(i); i += 1 } } })
```

Figure 6. Implementation of the Strm data type.

Marking the Strm class with an @embed annotation allows Squid to automatically create a deep embedding for the body of every method in the class (similar to [23]). By default, unless annotated with an explicit @phase (see Section 5.4), methods and data constructors are treated by Squid like syntactic sugar, and they are inlined on the fly.

Perhaps surprisingly, most of the constructs of our streams library already fuse automatically after inlining. For example, the “Hello World” of fusion `xs.map(f).map(g).sum` where `s.sum = s.fold(0)(_ + _)` basically desugars/inlines into:

```
val p = xs.producer(); var cur = 0
var cont = true; while (cont) { cont = false
  p { a => cur = cur + g(f(a)); cont = true } }
cur
```

As a more interesting example, consider the program:

```
optimize{ (xs:Array[Int]) => unfold(0)(i => Some(i,i+1))
  .zip(fromArray(xs).filter(_ % 2 == 0)).foreach(print) }
```

For which Squid produces this code, slightly reformatted:

```
(xs_0: Array[Int]) => {
  val len_1 = xs_0.length; var st_2 = 0; var i_3 = 0;
  var cont_4 = true; while (cont_4) { cont_4 = false;
    val x_5 = st_2; st_2 = x_5 + 1;
    var cont_6 = true; while (cont_6) { cont_6 = false;
      val iv_7 = i_3; if (iv_7 < len_1) {
        val x_8 = xs_0(iv_7);
        if(x_8 % 2 == 0){ print((x_5,x_8)); cont_4=true }
        else cont_6 = true;
        i_3 = iv_7 + 1; }}}}
}
```

As we can observe, all of the Strm abstractions have been removed and closures have disappeared, leaving behind a residual program made only of variables and loops. Normalization plays a major role in this regard: on the one hand, Squid relies on ANF (cf. Section 4.1) to streamline block structures and inline “one-shot” lambdas (lambdas applied only once [21]); on the other hand, user-defined online rewrite rules allow getting rid of intermediate abstractions — notice that the code above does not contain any Option despite the unfold interface making use of them. We do not describe such normalizations here for lack of space, but the ones that apply in this case transform `Some(x).isDefined` and `Some(x).get` into `true` and `x` respectively.

5.3 The problem with flatMap

The only construct that does not play well with this state of affairs is flatMap, which is due to its intrinsic higher-order nature. To understand this, consider one of its possible implementations, shown below:

```
def flatMap[B](f: A => Strm[B]): Strm[B] = Strm(() => {
  val p=producer(); var curBp=Option.empty[Producer[B]]
  k => {
    var consumed = false
    loopWhile {
      if (!curBp.isDefined)
        p { a => curBp = Some(f(a).producer()) }
      curBp.fold(false) { bs =>
        bs { b => k(b); consumed = true }
        if (!consumed) { curBp = None; true } else false
      }}}}
```

Syntax `loopWhile{...}` is the same as `while({...}){...}`, and `opt.fold(d)(f)` applies function `f` on the value contained in option `opt` or returns `d` if `opt` is not defined. The implementation proceeds by storing the current producer of `B` elements in variable `curBp`. Whenever the current producer runs out of elements (variable `consumed` is not set to true after calling `bs`), we set `curBp` to the next producer, which is obtained by applying `f` on the next element `a` of `p`.

The problem is that `curBp` is a variable that stores a function, preventing its inlining (remember that type `Producer[B]` is an alias for `Consumer[B] => Unit`). Notice that each time the value of `curBp` is reset, it captures a *different* value of `a` that is *not* available outside of the continuation passed to `p`. Even if we know the body of `f`, we cannot naively inline it at its use site, in `bs{ b => ... }`, because we would no longer have access to that `a`. As was noted before [6, 7, 11], these complications derive directly from the power and generality of flatMap. In the general case, for each element of the source stream, the function passed to flatMap could return streams of arbitrary shapes constructed at runtime, making it unfeasible for a compiler to fuse the code based solely on static information. However, in a significant proportion of stream programs used in practice (if not the vast majority), flatMap is used with more “well-behaved” functions, for example functions that always produce the same shape of streams for each source element (see Section 5.7). Furthermore, it is often possible to reorganize a program so that the result of any flatMap is consumed all at once (the “push-based” approach) instead of one element at a time, which allows us to avoid the inefficient pull-based implementation shown above. In the next section we explore that approach, and in Section 5.7 we describe a more general but more complex and slightly less efficient solution.

5.4 Enabling more fusion by QSR

Our goal is to defer the inlining of flatMap and the other Strm operations so that we get a chance to rewrite stream usage patterns in a way that removes the need for pulling from flatMap results. To achieve this, we annotate all core Strm methods (those that are not syntactic sugar) with @phase (“Low”) to delay their inlining. We then take inspiration from Kiselyov et al. [24], who leverage the fact that flatMap is no more problematic if we can consume its elements using internal iteration (push-based approach, cf. foreach) instead

of external iteration (pull-based). We introduce the notion of *pullable* streams for streams that can be efficiently used with external iteration. To mark streams that are pullable, we use a dummy “marker” method `pull[A](as: Strm[A]): Strm[A]` also annotated with `@phase("Low")`, which simply returns its argument unchanged.¹⁰ We make `fromRange` and `unfold` syntactic sugar that wrap their body with a call to `pull`, since their implementations are pullable, and we define the propagation rules seen in the first part of Figure 7. These rules “float out” the `pull` wrapper as long as the outer operation is also pullable.¹¹

The next step is to define rules that fold usages of the stream combinators in order to enable internal iteration. To simplify this step, we redefine `fold` and `foreach` in terms of a `doWhile` method that consumes the elements of a stream as long as its argument function returns true:

```
@phase("Low") def doWhile (f: A => Bool) = {
  val p = producer(); loopWhile {
    var cont = false; p { a => cont = f(a) }; cont }
}
```

The *Folding* rules in the second part of Figure 7 then reduce stream combinators that are applied to `doWhile`. The last two rules of Figure 7 dispatch the implementation of `zip` depending on which of its two arguments is pullable. Similar to [24], in this section we do not explicitly handle the case where neither is pullable. Function `doZip` is syntactic sugar for a specialized versions of `doWhile`:

```
def doZip[A,B](s: Strm[A], p: Producer[B])(f: (A,B) => Bool) =
  s.doWhile { a => var c = false; p { b => c = f(a,b) }; c }
```

Schematically, our optimizer is organized as follows:

- **Desugaring:** This is already done automatically by Squid; it concerns `fromArray`, `fold`, `foreach`, `doZip`, etc.
- **Flow:** bottom-up rewriting to propagate the `pull` information “down” the method chain — when possible — and to reduce consumed streams using internal iteration.
- **Lowering:** inlining of the `Strm` constructor, `pull`, `doWhile` and other core `Strm` methods to low-level code; removal of `Option` variables and other low-level optimizations.

Example. Consider the following pipeline transformation:

```
// Source:
fromRange(0,n) zip (
  fromRange(0,m).map(i => fromRange(0,i)).flatMap(x => x)
) filter {x => x._1 % 2 == 0} foreach println

// After Desugaring:
pullable(fromRangeImpl(0,n)).zip(
  pullable(fromRangeImpl(0,m))
    .map(i => pullable(fromRangeImpl(0,i)))
    .flatMap(x => x)
).filter { x => x._1 % 2 == 0 }
  .doWhile { x => println(x); true }
```

¹⁰ A common technique, also used by GHC developers. For example see <https://ghc.haskell.org/trac/ghc/wiki/OneShot> (accessed June 28 2017).

¹¹ Note that Squid allows type-parametric matching – unquotes can extract types as well as terms. For ease of presentation, Figure 7 has been simplified not to include type extraction. In our code base, the pattern of the first rule actually reads: `case code"pull[$ta]($as) map[$tb] $f" => ...`

```
@bottomUp @fixedPoint val Flow = rewrite {
  // Floating out pullable info
  case code"pull($as) map $f" => code"pull($as map $f)"
  case code"pull($as) filter $pred" => code"pull($as filter $pred)"
  case code"pull($as) take $n" => code"pull($as take $n)"
  case code"pull($as) flatMap $f" => code"$as flatMap $f"
  // Folding ^ flatMap is not 'pullable'
  case code"pull($as) doWhile $f" => code"$as doWhile $f"
  case code"$as map $f doWhile $g" => code"$as doWhile ($f andThen $g)"
  case code"$as filter $pred doWhile $f" => code"$as doWhile { a => !$pred(a) || $f(a) }"
  case code"$as take $n doWhile $f" => code""var tk = 0
    $as doWhile { a => tk += 1; tk <= $n && $f(a) }""
  case code"$as flatMap $f doWhile $g" => code""$as doWhile { a => var c = false
    $f(a) doWhile { b => c = $g(b); c }; c }""
  // Zipping
  case code"$as zip pull($bs) doWhile $f" => code""
    $as.doZip($bs.producer()){ (a,b) => $f((a,b)) }""
  case code"pull($as) zip $bs doWhile $f" => code""
    $bs.doZip($as.producer()){ (b,a) => $f((a,b)) }""
}
```

Figure 7. Algebraic rewrite rules for stream fusion.

```
// After Flow:
val p = fromRangeImpl(0,n).producer()
fromRangeImpl(0,m) doWhile { i =>
  var cont_0 = false
  fromRangeImpl(0,i) doWhile { b =>
    var cont_1 = false
    p { a => if(a % 2 == 0) println((a,b)); cont_1 = true }
    cont_0 = cont_1
    cont_0 }; cont_0 }
```

// After Lowering, the code has only variables and loops

To conclude this part, let us remark that we already fuse more programs than [24], because in contrast with that work we do not desugar `filter` to `flatMap`. The implementation of `filter` is pullable while that of `flatMap` is not, so that desugaring is counterproductive. As a result, we can fuse programs such as `(s0 filter f0) zip (s1 filter f1) while` [24] cannot — in their case writing such a program results in the generation of variables holding closures that capture local mutable state, a failure of abstraction removal.

By using a careful design, simple high-level rewrite rules and controlled inlining, we achieved state-of-the-art stream fusion capabilities with less complexity than previous work.

5.5 Correctness of the Stream Fusion scheme

The first desirable property for our stream fusion rewriting system is that it terminates, expressed in Theorem 5.1 below:

Theorem 5.1 (Strong Normalization). *The fixed point application of the rewrite rules in Figure 7 always converges.*

Proof: The `pull` wrapper propagation converges because `pull` is only propagated outwards, and is never introduced by any other rule. For the rest of the rules, notice that they each reduce the number of `Strm` constructs in the program

by at least one. This is a decreasing measure which ensures that the rewriting is terminating.

Next, let us argue that we fuse all stream pipelines that we set out to fuse (which excludes pipelines zipping two flattened streams). Interestingly, Figure 7 can be viewed like small-step operational semantics, where values are fully-fused stream pipeline programs. Our goal is then to show that well-formed pipelines reduce to values, which is done via the usual properties of *subject reduction* (Theorem 5.4) and *progress* (Theorem 5.6).

Definition 5.2 (Pullable Stream). A stream term that can be rewritten to a term of the form `pull(xs)` by the rewrite rules in Figure 7.

Definition 5.3 (Well-formed pipeline). A well-typed program where: 1. all stream sub-terms are used as arguments in applications of `map`, `flatMap`, `filter`, `take`, `zip`, `pull`, or `doWhile`; 2. where any applications of `zip` has at least one of its two arguments pullable; and 3. where applications of `pull` enclose neither `zip` nor `flatMap` applications.

Notice that in actual programs, `pull` is not invoked by the user but solely arises from desugaring `fromRange(n,m)` and `unfold(z)(f)` into well-formed sub-terms, respectively `pull(fromRangeImpl(n,m))` and `pull(unfoldImpl(z)(f))`.

For simplicity we consider that `producer()` calls introduced by the *Zipping* rules are immediately inlined, and that the resulting code is inlined as well, recursively.

Theorem 5.4 (Subject Reduction). *The rewrite rules of Figure 7 preserve types and well-formedness.*

Proof: We have type preservation for free thanks to Squid, which statically enforces that the result of each rewrite rule has the same type as the pattern. It is straightforward to see that well-formedness is preserved as well, as the rules only introduce `Strm` functions and low-level constructs — we can prove by induction that `producer()` is never applied on `flatMap` (which is the only construct with a non-trivial `producer` implementation) because the zipping rules make sure `producer()` is only applied on pullable streams, and `flatMap` does not propagate the `pull` wrapper.

Definition 5.5 (Fully-fused pipeline). A program that only contains references to `doWhile`, `fromRangeImpl`, `unfoldImpl` and low-level constructs such as variables of primitive types, if-then-else branches, conditionals, etc.

Theorem 5.6 (Progress). *Any well-formed pipeline that is not fully-fused can have at least one of its sub-terms reduced by the rules of Figure 7.*

Proof: For a pipeline to be well-formed but not fully fused, it either needs to have non-low-level features such as function variables — which cannot happen because we only inline pullable streams — or it needs to still have one of `pull`, `map`, `flatMap`, `filter`, `take` or `zip`. At least one of these has to be passed into a call to `doWhile`, by well-formedness hypothesis (because `doWhile` is the only terminal operation). Therefore, such term can be reduced by one of the folding rewrite rules

if the outer term is not a `zip`. If it is a `zip`, we can either propagate `pull` in one of its arguments, or we can apply one of the zipping rules because by hypothesis at least one of the two arguments is pullable.

Finally, remark that *semantic* preservation (in terms of program execution semantics) is easily assessed by looking at each rewrite rule case in isolation. In other words, QSR makes it easy to reason locally about each rewrite rule, ensuring that it transforms its input program into an equivalent output.

5.6 Extensibility of Optimizations

We already saw that Squid allows adding syntactic sugar in the form of user-defined methods annotated with or enclosed by a class annotated with `@embed` (which allows Squid to lift the method's implementation). Here we examine how to extend the set of core stream constructs.

As an example, consider stream programs that contain code of the form `if (...) stream0 else stream1`. At the moment, that pattern will not be recognized as pullable by the library, and may therefore get in the way of fusion. Thankfully, by only adding the two rules below we can seamlessly integrate that pattern with the rest of the fusion system:

```
// Floating out pullable info
case code "if ($c) pull($thn) else pull($els)"
=> code "pull(if ($c) $thn else $els)"
// Folding
case code "( if($c) $thn else $els ) doWhile $f"
=> code "" "if ($c) $thn doWhile inl($f)
           else $els doWhile inl($f)"""
```

Like `pull(s)`, syntax `inl(f)` is used as a marker. It hints for Squid to inline function `f`, even if `f` is used in several places. This effectively leads to code duplication in the case above, but that is a requirement for fusion to happen reliably.

Contrast the seamless extension above with what [24] proposes to solve the same problem, which involves *changing the user interface* of `flatMap` to continuation-passing style.

5.7 When everything else fails — streamlining `flatMap` the hard way

The rewriting proposed in Section 5.4 generates fused code for many use cases, but unfortunately fails to fuse `zip` applications where both arguments are flattened. More generally, it fails to fuse `flatMap` in the absence of a direct consumer of the flattened stream, which can also happen if we only have access to incomplete pipelines, such as `(n: Int) => fromRange(0, n).flatMap(fromRange(0, _))`. In general, it would be useful if we could *streamline* `flatMap` applications so as to make them efficiently pullable. Intuitively, the code above could be rewritten:

```
(n: Int) => Strm() => { var i = 0; var j = None; k =>
  loopWhile { // loop until suitable element is found
    if (j.isEmpty && i < n) { j = Some(i); i += 1 }
    j.fold(false) { jv =>
      if (jv < i) { k(jv); false } // element found
      else { j = None; true } } }) // j stream exhausted
```

Notice the similarity with the implementation of `flatMap` shown in Section 5.3. The main difference is that instead of using a variable that stores the inner `Producer`, we use a variable that stores the *state* of the inner producer (here

```

@online val FlatMapStreamlining = rewrite {
case code"doFlatMap[$ta,$tb]($pa,a => $f(a))" => close(f){
(bodyf, reopenf) =>
def rec(t: Code[Producer[tb]], reset: Code[() => Unit])
: Code[Producer[tb]] = t match {
case code"val x = Var[$xt]($init); $innerBody(x)"
=> close(innerBody) { (ib,rib) =>
code"val y = Var(null : $xt); ${ (y:Code[Var[xt]]) =>
val newReset = code"() => { $reset(); $y := $init }"
rib(rec(ib, newReset))(y)
}(y)" }
case code"val x: $xt = $init; $innerBody(x)"
=> close(innerBody) { (ib,rib) =>
code"val y = Var(null : $xt); ${ (y:Code[Var[xt]]) =>
val newReset = code"() => { $reset(); $y := $init }"
rib(rec(ib, newReset))(code"$y.!" )
}(y)" }
case code"$effect; $innerBody"
=> rec(innerBody, code"() => { $reset(); $effect }")
case code"k => $innerBody(k)" => code""
var curA: Option[A] = None
(k:Consumer[tb]) => { var consumed = false; loopWhile {
if (!curA.isDefined)
$P { a => curA = Some(a); ${reopenf(reset)}(a)() }
curA.fold(false) { a =>
${reopenf(t)}(a) { b => k(b); consumed = true }
if (!consumed) { curA = None; true } else false }}}
""
case _ => throw StagingError("Could not streamline") }
rec(bodyf, code"() => ()")
}}

```

Figure 8. The flatMap streamlining rewrite rule.

j), and that state is reset whenever a new value of the outer producer (here *i*) is obtained. The idea is similar to the one proposed in [11], though we use actual imperative stream states while [11] uses a purely functional encoding of stream states, as the host language (Haskell) is purely functional.

This transformation can be applied automatically, provided we have access to the complete inlined state of the inner producer. This tells us that the rewriting should apply after the *Lowering* phase of Section 5.4. To prevent flatMap from being inlined to its inefficient implementation, we change its implementation so that it inlines into a low-level doFlatMap(*p*, *f*) method where *p* is the source producer, and *f* is the function that creates an inner producer from each element of *p*. Next, we register the rewrite rule of Figure 8, to be applied during *Lowering*. The code in Figure 8 is the most technical example of our paper; to understand it, we first need to introduce a few concepts:

Higher Order Patterns Variables. Squid provides a very simple form of higher-order matching [8, 32], that directly mirrors the automatic function lifting facility. Concretely, while pattern `code"(x:Int) => $body:Int"` will not match a lambda where *body* makes use of *x*, the following pattern will: `code"(x:Int) => $f(x):Int"`, giving to *f* type `Code[Int] => Code[Int]`. Applying *f* to some `Code[Int]` will replace all usages that *f* made of *x* with the provided code value. Higher order pattern variables in quasiquote-based matching was suggested before by Sheard et al. [40].

Temporary Variable Extrusion. In order to inspect open code, which is represented as a function [32], we must apply it to some value first. However, sometimes we do not yet have that value until after we have inspected the code. To solve

this, Squid provides a `close(f)((body, reopen) => ...)` idiom used to temporarily close open code *f* as body, making it inspectable. Given some *f* : `Code[A] => Code[B]`, the type of *body* is *B* but it implicitly contains unbound references to its *A* parameter. *reopen* has type $\forall x. \text{Code}[X] \Rightarrow \text{Code}[A \Rightarrow X]$, and is used to reintroduce the explicit parameter dependency. *reopen* can be applied to *body* or to any of its subterms. This mechanism can lead to errors, if one does not *reopen* pieces of code that were closed and contain occurrences of the parameter. However, the `close` function checks that no such occurrences exist in the result. Therefore, any programmer error is immediately reported at the relevant place, and we never end up with programs containing unbound or wrongly-captured variables, which greatly simplifies debugging.

Virtualized mutable variables. To facilitate the handling of mutable variable bindings, Squid views all mutable variables as syntactic sugar for usages of the `Var[T]` data type, an approach inspired by language virtualization [28]. `Var[T]` features methods `!` and `:=` to respectively access the value of a variable and reset it. Therefore `code"var i = 0; i += 1"` is equivalent to `code"val i = Var(0); i := i.!. + 1"`.

In Figure 8 we recursively analyse the body of *f*, using a `reset` parameter to accumulate a function term that resets the state of the inner stream when a new element of the outer stream is available. When we encounter a mutable variable binding, we reconstruct it but initialize it with `null` and integrate the actual initialization as part of the `reset` argument passed recursively. Immutable value bindings are converted into variables so they can be reset similarly. Finally, encountered effects are simply integrated into the `reset` accumulator. When the recursion finally encounters the `k => ...` lambda expression constructing the resulting `Producer`, we build the final, efficient implementation of `flatMap`. Note that it is important to match a lambda `k => ...` term, for the soundness of the rewriting (finding something else in place of a lambda would mean that low-level state inlining might not have been complete).

Summary. We presented an algorithmic rewrite rule that performs flatMap streamlining to enable fusion in many of the cases where it failed in Section 5.4. The rule is implemented entirely using the type-safe, high-level utilities provided by Squid. Previously, Farmer et al. [11] demonstrated a similar rewriting, but to implement it they had to extend the compiler and drop down to complicated IR manipulations. We can also argue that our approach is more robust, as the user has full control on the rewriting and inlining pipeline. Remark that flatMap streamlining usefully completes the scheme of Section 5.4, but does not replace it: our optimizer always tries to apply the latter first, because it is more efficient (as it deals with smaller, higher-level stream representations) and because it tends to produce slightly better code with less variables and loops.

5.8 Conclusion

Quoted Staged Rewriting allows library writers to co-design high-level libraries and associated optimizers that render their usage as efficient as low-level code (cf. Section 6). Appendix A details how the `strm` library is organized.

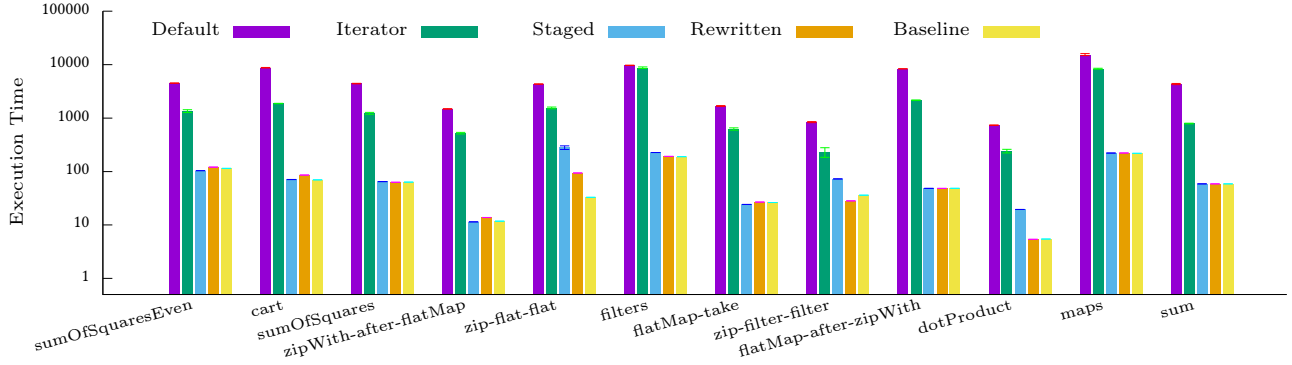


Figure 9. Time taken by different stream pipeline implementations on the JVM. Notice the logarithmic scale. Default: our streams library without optimization; Iterator: standard Scala iterators; Staged: staged streams using LMS [24]; Rewritten: our streams library with optimizations applied; Baseline: manual low-level implementations. The code is available online at <https://github.com/epfldata/staged-rewritten-streams>.

6 Experiments

Performance. In this section, we empirically demonstrate that our QSR stream fusion approach¹² is competitive with both staging and manual low-level implementations that use only integral variables and loops. We measured the execution time of small pipelines consisting of flat-mapping, filtering, zipping, etc. and summing up the results. We tested five approaches: our pure Scala library without optimization (Default); Scala iterators, which are conceptually similar but lower level (their interface is imperative) and hand-optimized to play well with the JVM (Iterator); the staged code of [24] (Staged); the same code as Default but surrounded with an `optimize{...}` block to apply our QSR (Rewritten); and hand-coded low level implementations using integral variables and loops (Baseline). The inputs used consisted of arrays of several hundred thousand integer elements. The times were measured on a six-core Intel Xeon E5-2620 v2 processor with 256GB of DDR3 RAM (1600Mhz). We used Scala version 2.11.2 running on the OpenJDK 64-Bit Server VM (build 24.95-b01) with Java 1.7.0_101.

As we can observe, the rewritten version has performance characteristics mostly similar to the staged and low-level versions. All these three versions outperform the unoptimized and iterator versions by one or two orders of magnitude. We interpret that performance difference as the cost of abstraction (here mainly incurred from using closures, virtual dispatch and boxing). Even in these simple cases the advanced JIT of the JVM in server mode cannot remove that overhead automatically. Both the rewritten and staged approaches produce similarly low-level code where all abstractions have been eliminated, except for cases `zip_flat_flat` and `zip_filter_filter` where the staged version fails to fuse and falls back to using variables holding functions. The generated code still performs honorably — about twice as slow as the rewritten version but an order of magnitude faster than the unoptimized ones. Finally, notice that even if the rewritten version does completely fuse `zip_flat_flat` using the technique in Section 5.7, the resulting code is still significantly slower than the baseline, as we produce more variables and more intricate loops. For the rest of the tests,

¹² We benchmark the fusion algorithm presented in Section 5, with a few minor tweaks and extra normalization rules that help with streamlining.

Table 1. Lines of code for stream fusion in Squid and LMS.

	Shallow	Fusion	Generic
Squid / QSR	127	149	293
LMS / [24]	—	314	1982

the minor differences in runtime between the staged, rewritten and baseline versions can be attributed to slight differences in generated looping structures.

Productivity. We conclude with a brief empirical argument about the productivity gains of our approach. We measured the number of physical lines of code (i.e., excluding comments and blank lines) in: 1. the “shallow” implementation of the library (cf. Figure 6); 2. the implementation of stream fusion; and 3. the supporting library code that allows the approaches to function (*Generic*). The LMS implementation does not have a shallow counterpart to its staged streams library — which is reported under *Fusion*. The *Generic* number for LMS accounts for the IR definitions of basic constructs such as arrays, strings, tuples, etc., and associated code generation implementations; it only includes LMS code used by this application. For Squid, *Generic* includes mainly standard normalization rules as well as a small library of virtualized constructs (like mutable variables). Notice that the stream fusion rewritings in Squid are half the size of the staging-based implementation in LMS, and are completely separate from the library, which can be used independently. Even more tellingly, the LMS approach requires considerably more supporting code,¹³ and that code will only grow as users want to include more constructs to be used within their streams programs. In contrast, Squid accommodates new constructs without requiring any additional supporting code.

Acknowledgments

We would like to thank Aggelos Biboudis for his constructive input on the paper, and we are grateful to the reviewers for their useful comments. The `optimize{...}` macro syntax was inspired by ScalaBlitz.¹⁴ This work was supported by NCCR MARVEL of the Swiss National Science Foundation.

¹³ This code can be generated automatically by tools like Yin-Yang [23] and Forge [43], but it is still an overhead compared to not needing it at all.

¹⁴ <https://web.archive.org/web/20141218200210/scala-blitz.github.io/>

References

- [1] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. 2010. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE)*, 2010 8th IEEE/ACM International Conference on. IEEE, 169–178.
- [2] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 05 (2009), 509–543.
- [3] James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-integrated Query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 403–416.
- [4] Cliff Click. 1995. Global Code Motion/Global Value Numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 246–257.
- [5] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *TOPLAS* 17, 2 (March 1995), 181–196.
- [6] Duncan Coutts. 2011. *Stream fusion : practical shortcut fusion for coinductive sequence types*. Ph.D. Dissertation. University of Oxford, UK.
- [7] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion. From Lists to Streams to Nothing at All. In *ICFP '07*.
- [8] Oege de Moor and Ganesh Sittampalam. 2001. Higher-order matching for program transformation. *Theoretical Computer Science* 269, 1-2 (2001), 135–162.
- [9] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 105–116.
- [10] Andrew Farmer. 2015. *HERMIT: Mechanized Reasoning during Compilation in the Glasgow Haskell Compiler*. Ph.D. Dissertation. University of Kansas.
- [11] Andrew Farmer, Christian Hoener zu Siederdisen, and Andy Gill. 2014. The HERMIT in the stream: fusing stream fusion's concatMap. In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation*. ACM, 97–108.
- [12] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *ACM Sigplan Notices*, Vol. 28. ACM, 237–247.
- [13] Steven E Ganz, Amr Sabry, and Walid Taha. 2001. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 74–85.
- [14] Andrew Gill, John Launchbury, and Simon L Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*. ACM, 223–232.
- [15] Andrew John Gill. 1996. *Cheap deforestation for non-strict functional languages*. Ph.D. Dissertation. University of Glasgow.
- [16] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *Proceedings of the 7th international conference on Generative programming and component engineering*. ACM, 137–148.
- [17] Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)* 28, 4es (1996), 196.
- [18] Dean F Jerding, John T Skasko, and Thomas Ball. 1997. Visualizing interactions in program executions. In *Proceedings of the 19th international conference on Software engineering*. ACM, 360–370.
- [19] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.
- [20] Simon L. Peyton Jones. 1996. *Compiling Haskell by program transformation: A report from the trenches*. Springer Berlin Heidelberg, Berlin, Heidelberg, 18–44.
- [21] Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. [n. d.]. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop*. ACM SIGPLAN.
- [22] Manohar Jonnalagedda and Sandro Stucki. 2015. Fold-based Fusion As a Library: A Generative Programming Pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala (SCALA 2015)*. ACM, 41–50.
- [23] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-Yang: Concealing the Deep Embedding of DSLs (*GPCE 2014*). ACM, 73–82.
- [24] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinis, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 285–299.
- [25] P. Klint, T. v. d. Storm, and J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 168–177.
- [26] HyoukJoong Lee, Kevin J Brown, Arvind K Sajeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro* 31, 5 (2011), 42–53.
- [27] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. 2015. Shallow Embedding of DSLs via Online Partial Evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*. ACM, New York, NY, USA, 11–20.
- [28] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*. ACM, 117–120.
- [29] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old is New Again: Quoted Domain-specific Languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2016)*. ACM, New York, NY, USA, 25–36.
- [30] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE '13)*. ACM, New York, NY, USA, 125–134.
- [31] Lionel Parreaux, Amir Shaikhha, and Christoph Koch. 2017. Squid: Type-Safe, Hygienic, and Reusable Quasiquotes. In *Proceedings of the 2017 8th ACM SIGPLAN Symposium on Scala (SCALA 2017)*. ACM.
- [32] Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, Vol. 23. ACM, 199–208.
- [33] Markus Püschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.
- [34] Tiark Rompf. 2016. Reflections on LMS: exploring front-end alternatives. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. ACM, 41–50.
- [35] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming and Component Engineering*. 127–136.
- [36] Tiark Rompf, Arvind K. Sajeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*. 497–510.
- [37] Tiark Rompf, Arvind K Sajeeth, HyoukJoong Lee, Kevin J Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-blocks for performance oriented DSLs. *DSL* (2011).
- [38] Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *ECOOP*. Springer, 258–282.

- [39] Maximilian Scherr and Shigeru Chiba. 2015. Almost First-class Language Embedding: Taming Staged Embedded DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*. ACM, New York, NY, USA, 21–30.
- [40] Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic. 1999. DSL Implementation Using Staging and Monads. In *Proceedings of the 2Nd Conference on Domain-specific Languages (DSL '99)*. ACM, New York, NY, USA, 81–94.
- [41] Anthony M. Sloane. 2011. Lightweight Language Processing in Kiama. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III (GTTSE'09)*. Springer-Verlag, Berlin, Heidelberg, 408–425.
- [42] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. *ACM SIGPLAN Notices* 50, 9 (2015), 205–217.
- [43] Arvind K Sujeeth, Austin Gibbons, Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: Generating a high performance DSL implementation from a declarative specification. In *Proceedings of the 12th international conference on Generative programming*. ACM, 145–154.
- [44] Walid Taha. 1999. *Multi-stage programming: Its theory and applications*. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.
- [45] Walid Taha. 2004. *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter A Gentle Introduction to Multi-stage Programming, 30–50.
- [46] Walid Taha and Tim Sheard. 1997. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 203–217.
- [47] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242.
- [48] Vlad Ureche, Aggelos Biboudis, Yannis Smaragdakis, and Martin Odersky. 2015. Automating Ad Hoc Data Representation Transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 801–820.
- [49] Eelco Visser. 2001. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science* 57 (2001), 109–143.
- [50] Eelco Visser. 2002. Meta-programming with concrete object syntax. In *Generative programming and component engineering*. Springer, 299–315.
- [51] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building program optimizers with rewriting strategies. In *ACM Sigplan Notices*, Vol. 34. ACM, 13–26.
- [52] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building Program Optimizers with Rewriting Strategies (*ICFP '98*). 13–26.
- [53] Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*. Springer, 344–358.
- [54] Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84.
- [55] Thomas Würthinger. 2011. Extending the graal compiler to optimize libraries. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 41–42.
- [56] Jeremy Yallop and Leo White. 2015. Modular macros. In *OCaml Users and Developers Workshop*.

A Organization of the Streams Optimizer

In this appendix, we give more details on the way a library should be structured in order to benefit from Squid-based Quoted Staged Rewriting, using the streams library seen in Section 5 as an example.

Scala Restrictions. Squid makes extensive use of macros, which currently have some restrictions: Scala code defined in some project P cannot be executed at compile-time in P itself or in a project that P depends on — one may have to “stratify” program definitions into different sub-projects.

The Strm Library. It is organized in a single project as follows: in the `lib` package, the shallow `Strm` definitions as seen in Figure 6, with an `@embed` annotation to automatically lift method definitions; in the `compiler` package, the ANF-based “embedding” used as the IR in which to manipulate the code (Squid supports different IRs [31]), defined as follows:

```
object Embedding extends SchedulingANF
  with OnlineOptimizer with StandardEffects
{ object Desug extends Transformer with Desugaring
  object Norm extends Transformer
    with StandardNormalizer with LogicNormalizer
  def pipeline = Desug.pipeline andThen Norm.pipeline
  embed(Strm)
}
```

Object Embedding extends the `SchedulingANF` base IR and the `StandardEffects` trait to benefit from effect annotations on standard Scala constructs. It extends `OnlineOptimizer` in order to perform some online rewriting — provided via the `pipeline` method, which applies some desugaring and then some normalization. The `embed(Strm)` call, executed when Embedding is initialized, registers in this IR the `Strm` methods lifted earlier by `@embed`.

The stream fusion optimizer itself is implemented in the `compiler.StrmOptimizer` class, which defines successive optimization phases `Flow`, `Lowering` and `LowLevel`:

```
class StrmOptimizer extends Optimizer
{ def pipeline = (
  Flow.pipeline
  andThen Lowering.pipeline
  andThen LowLevel.pipeline )
}
object Flow extends Embedding.Transformer
  with SimpleRuleBasedTransformer
  with BottomUpTransformer
  with FixPointTransformer
{ rewrite { ... } }
object Lowering extends Embedding.Transformer with ...
object LowLevel extends Embedding.Transformer with ...
```

The way `Flow` is defined above is equivalent to the annotation-based way seen in the paper, which is only syntax sugar (i.e., `@bottomUp @fixedPoint val Flow = rewrite{ ... }`). The role of `LowLevel` is to apply low-level transformations at the end of the pipeline, such as flattening variables holding an option type into a boolean variable `isDefined` and an unwrapped `currentValue` variable.

In order to use this optimizer, one has to instantiate class `StaticOptimizer[Strm.compiler.StrmOptimizer]` and, from another project, invoke its `optimize{ ... }` macro.